ECE421 Introduction to Machine Learning
Assignment #2
Neural Networks

Yan, Xuanming
[redacted]
Contribution: 50%

Wang, Chu Qing
[redacted]
Contribution: 50%

Feb 2019

1 Neural Networks using Numpy

1.1 Helper Functions

1. *ReLU()*: $ReLU(x) = max(x, 0)$

```
def relu(x):
    relu_x = np.maximum(x, 0)
    return relu_x
```

Listing 1: ReLU()

2. *softmax()*: $\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1 \dots K$, for K classes.

```
def softmax(x):
    softmax_x = np.exp(x)/sum(np.exp(x))
    return softmax_x
```

Listing 2: softmax()

3. *compute()*:

```
def computeLayer(X, W, b):
    compute_layer = np.matmul(X_trans, W) + b
    return compute_layer
```

Listing 3: compute()

4. *averageCE()*: $CE = -\frac{1}{N}\sum_{k=1}^{K} t_k log(s_k)$

```
def CE(target, prediction):
    ce = -np.mean(target*np.log(prediction))
    return ce
```

Listing 4: averageCE()

5. *gradCE()*: $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, $\mathbf{s} = softmax(\mathbf{z}) = \frac{e^{\mathbf{z}}}{\sum_{k=1}^{K} e^{z_k}}$,

$$\begin{aligned}
\mathcal{L}_{CE} &= -\sum_{k=1}^{K} t_k log(s_k) \\
&= -\mathbf{t}^T(log(\mathbf{s})) \\
\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{s}} &= -\mathbf{t}^T(\frac{\partial}{\partial \mathbf{s}}(log(\mathbf{s}))) \\
&= -\mathbf{t}^T\frac{1}{\mathbf{s}}
\end{aligned}$$

Derivative of softmax can be derived:

$$\begin{aligned}\frac{\partial \mathbf{s}}{\partial \mathbf{z}} &= \frac{e^{\mathbf{z}}(\sum_{k=1}^{K} e^{z_k}) - (e^{\mathbf{z}})^T e^{\mathbf{z}}}{(\sum_{k=1}^{K} e^{z_k})^2} \\ &= \frac{e^{\mathbf{z}}}{\sum_{k=1}^{K} e^{z_k}} \frac{\sum_{k=1}^{K} e^{z_k} - e^{\mathbf{z}}}{\sum_{k=1}^{K} e^{z_k}} \\ &= \mathbf{s}(1-\mathbf{s})\end{aligned}$$

Apply chain rule, we can get:

$$\begin{aligned}\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{z}} &= \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{z}} \\ &= (-\mathbf{t}^T \frac{1}{\mathbf{s}})(\mathbf{s}(1-\mathbf{s})) \\ &= \mathbf{s} - \mathbf{t}\end{aligned}$$

```
def gradCE(target, prediction):
    softmax_ce = prediction - target
    return softmax_ce
```

Listing 5: gradCE()

1.2 Back-propagation Derivation

Suppose we have **N** inputs:

1. $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_o}$, the gradient of the loss with respect to the outer layer weights. Shape: $(K \times 10)$, with K units. $\mathbf{W}_o \in \mathbb{R}^{K \times 10}$, since output layer has activation function of Softmax, we have:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} &= \frac{\partial \mathcal{L}}{\partial \mathbf{s}_o} \frac{\partial \mathbf{s}_o}{\partial \mathbf{z}_o} = \frac{\partial \mathcal{L}}{\partial \mathbf{s}_o} \mathbf{s}_o(1-\mathbf{s}_o) \\ \frac{\partial \mathcal{L}}{\partial \mathbf{s}_o} &= -\mathbf{y}_{true} \frac{1}{\mathbf{s}_o} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} &= \frac{\partial \mathcal{L}}{\partial \mathbf{s}_o} \frac{\partial \mathbf{s}_o}{\partial \mathbf{z}_o} = (-\mathbf{y}_{true} \frac{1}{\mathbf{s}_o})(\mathbf{s}_o(1-\mathbf{s}_o)) = \mathbf{s}_o - \mathbf{y}_{true}\end{aligned}$$

   Where $\mathbf{z}_o$ and $\mathbf{s}_o$ are input and output of output layer, respectively. $\mathbf{z}_o \in \mathbb{R}^{N \times 10}$, and $\mathbf{s}_o \in \mathbb{R}^{N \times 10}$. We also have $\mathbf{z}_o = \mathbf{s}_h \mathbf{W}_o + \mathbf{b}_o$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_o} = \frac{\partial \mathbf{z}_o}{\partial \mathbf{W}_o} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} = \mathbf{s}_h^T(\mathbf{s}_o - \mathbf{y}_{true})$$

   Here, $\mathbf{s}_h \in \mathbb{R}^{N \times K}$ is the output of previous hidden layer.

```
def back_out_weight(target, prediction, hidden_out):
    softmax_ce = gradCE(target, prediction)
    hidden_out_transpose = np.transpose(hidden_out)
    grad_out_weight = np.matmul(hidden_out_transpose, softmax_ce)
    return grad_out_weight
```

Listing 6: back prop 1

2. $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_o}$, the gradient of the loss with respect to the outer layer biases. Shape:(1×10). We can easily derive $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_o}$ as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_o} = \frac{\partial \mathbf{z}_o}{\partial \mathbf{b}_o} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} = \mathbf{1}^T(\mathbf{s}_o - \mathbf{t})$$

   Here, $\mathbf{1}$ is a vector which all elements are one. $\mathbf{1} \in \mathbb{R}^{N \times 1}$.

```
def back_out_bias(target, prediction):
    softmax_ce = gradCE(target, prediction)
    ones = np.ones((1, target.shape[0]))
    grad_out_bias = np.matmul(ones, softmax_ce)
    return grad_out_bias
```

Listing 7: back prop 2

3. $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h}$, the gradient of the loss with respect to the hidden layer weights. Shape: $(F \times K)$, with F features, K units. $\mathbf{W}_h \in \mathbb{R}^{F \times K}$, since hidden layer has activation function of ReLU, and $\mathbf{z}_h = \mathbf{s}_i \mathbf{W}_h + \mathbf{b}_h$, we have:

$$\frac{\partial \mathbf{s}_h}{\partial \mathbf{z}_h} = \begin{cases} 0, & \text{if } z_{hi} < 0 \\ 1, & \text{else if } z_{hi} > 0 \end{cases}$$

$$\frac{\partial \mathbf{z}_0}{\partial \mathbf{s}_h} = \mathbf{W}_o^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \frac{\partial \mathbf{z}_h}{\partial \mathbf{W}_h} \frac{\partial \mathbf{s}_h}{\partial \mathbf{z}_h} \frac{\partial \mathbf{z}_0}{\partial \mathbf{s}_h} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} = \mathbf{s}_i^T \bigotimes \frac{\partial \mathbf{s}_h}{\partial \mathbf{z}_h} (\mathbf{s}_o - \mathbf{y}_{true}) \mathbf{W}_o^T$$

Where $\mathbf{z}_h$ and $\mathbf{s}_h$ are input and output of hidden layer, respectively. $\mathbf{z}_h \in \mathbb{R}^{N \times K}$, and $\mathbf{s}_h \in \mathbb{R}^{N \times K}$, $\mathbf{s}_i \in \mathbb{R}^{N \times F}$ is input vector of this neural network.

```
def back_hidden_weight(target, prediction, input, input_out, out_weight):
    input_out[input_out > 0] = 1
    input_out[input_out < 0] = 0
    softmax_ce = gradCE(target, prediction)
    grad_hidden_weight = np.matmul(np.transpose(input), \
     (input_out * np.matmul(softmax_ce, np.transpose(out_weight))))
    return grad_hidden_weight
```

Listing 8: back prop 3

4. $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_h}$, the gradient of the loss with respect to the hidden layer biases. Shape: $(1 \times K)$, with K units.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} &= \frac{\partial \mathbf{z}_h}{\partial \mathbf{b}_h} \frac{\partial \mathbf{s}_h}{\partial \mathbf{z}_h} \frac{\partial \mathbf{z}_0}{\partial \mathbf{s}_h} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_o} \\ &= \mathbf{1}^T \bigotimes \frac{\partial \mathbf{s}_h}{\partial \mathbf{z}_h} (\mathbf{s}_o - \mathbf{y}_{true}) \mathbf{W}_o^T \end{aligned}$$

Here, $\mathbf{1}$ is a vector which all elements are one. $\mathbf{1} \in \mathbb{R}^{F \times 1}$

```
def back_hidden_bias(target, prediction, input_out, out_weight):
    input_out[input_out > 0] = 1
    input_out[input_out < 0] = 0
    ones = np.ones((1, input_out.shape[0]))
    softmax_ce = gradCE(target, prediction)
    grad_hidden_bias = np.matmul(ones, \
     (input_out * np.matmul(softmax_ce, np.transpose(out_weight))))
    return grad_hidden_bias
```

Listing 9: back prop 4

1.3 Learning

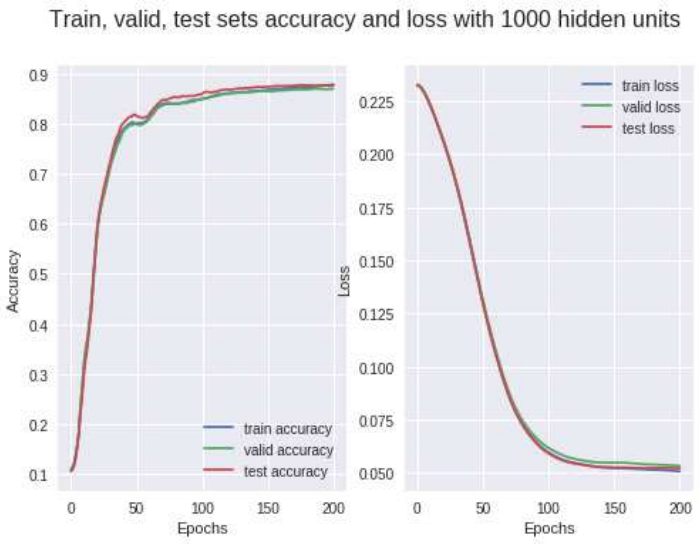

Figure 1: Training, validation test set losses and accuracy.

In this task, we construct a neural network with hidden unit size of 1000. First, we initialize our weight matrices using the Xaiver initialization scheme. The hidden layer weight has zero-mean Gaussian distribution with variance of $\frac{1}{728+1000}$. And output layer weight has zero-mean Gaussian distribution with variance of $\frac{1}{1000+10}$. And bias matrices are assigned to zeros. Learning rate for this network is 0.0000001, and $\gamma = 0.99$. The final accuracy we got are:

Training data accuracy	Valid data accuracy	Test data accuracy
88.06%	87.45%	88.18%

Table 1: Training, validation and test accuracy after training.

And this training took 323.45 seconds.

```
def learning(trainData, target, W_o, v_o, W_h, v_h, epochs, \
        gamma, learning_rate, bias_o, bias_h, validData, newvalid, testData, newtest):

    W_v_o = v_o
    b_v_o = bias_o
    W_v_h = v_h
    b_v_h = bias_h
    accuracy = []
    accuracy_valid = []
    accuracy_test = []
    loss = []
    loss_valid = []
    loss_test = []

    for i in range(epochs):

        hidden_input = np.add(np.matmul(trainData, W_h), bias_h)
        hidden_out = relu(hidden_input)
        prediction = softmax(np.add(np.matmul(hidden_out, W_o), bias_o))
        loss.append(CE(target, prediction))
        predict_result_matrix = np.argmax(prediction, axis = 1)
        actural_result_matrix = np.argmax(target, axis = 1)
        compare = np.equal(predict_result_matrix, actural_result_matrix)
        accuracy.append(np.sum((compare==True))/(trainData.shape[0]))


        hidden_input_valid = np.add(np.matmul(validData, W_h), bias_h)
        hidden_out_valid = relu(hidden_input_valid)
        prediction_valid = softmax(np.add(np.matmul(hidden_out_valid, W_o), bias_o))
        loss_valid.append(CE(newvalid, prediction_valid))
        predict_result_matrix_valid = np.argmax(prediction_valid, axis = 1)
        actural_result_matrix_valid = np.argmax(newvalid, axis = 1)
        compare_valid = np.equal(predict_result_matrix_valid, actural_result_matrix_valid)
        accuracy_valid.append(np.sum((compare_valid==True))/(validData.shape[0]))


        hidden_input_test = np.add(np.matmul(testData, W_h), bias_h)
        hidden_out_test = relu(hidden_input_test)
        prediction_test = softmax(np.add(np.matmul(hidden_out_test, W_o), bias_o))
        loss_test.append(CE(newtest, prediction_test))
        predict_result_matrix_test = np.argmax(prediction_test, axis = 1)
        actural_result_matrix_test = np.argmax(newtest, axis = 1)
        compare_test = np.equal(predict_result_matrix_test, actural_result_matrix_test)
        accuracy_test.append(np.sum((compare_test==True))/(testData.shape[0]))

        print("Iteration:", i)
        W_v_o = gamma*W_v_o + learning_rate*back_out_weight(target, prediction, hidden_out)
        W_o = W_o - W_v_o
        b_v_o = gamma*b_v_o + learning_rate*back_out_bias(target, prediction)
        bias_o = bias_o - b_v_o

        W_v_h = gamma*W_v_h + learning_rate*back_hidden_weight(target, \
            prediction, trainData, hidden_input, W_o)
        W_h = W_h - W_v_h
        b_v_h = gamma*b_v_h + learning_rate*back_hidden_bias(target, prediction, hidden_input, W_o)
        bias_h = bias_h - b_v_h
        # print("prediction: ", W_o)

    return W_o, bias_o, W_h, bias_h, accuracy, accuracy_valid, accuracy_test, loss, loss_valid,
    loss_test

```

```
61
62 if __name__ == '__main__':
63     trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
64     trainData = trainData.reshape((trainData.shape[0], trainData.shape[1]*trainData.shape[2]))
65     validData = validData.reshape((-1,validData.shape[1]*validData.shape[2]))
66     testData = testData.reshape((-1,testData.shape[1]*testData.shape[2]))
67
68     hidden_units = 1000
69     epochs = 200
70     gamma = 0.99
71     learning_rate = 0.0000001
72
73
74     newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)
75     mu = 0 # mean and standard deviation
76     stddev_o = 1.0/(hidden_units+10)
77     W_o = np.random.normal(mu, np.sqrt(stddev_o), (hidden_units,10))
78     v_o = np.full((hidden_units, 10), 1e-5)
79
80     stddev_h = 1.0/(trainData.shape[1]+hidden_units)
81     W_h = np.random.normal(mu, np.sqrt(stddev_h), (trainData.shape[1],hidden_units))
82     v_h = np.full((trainData.shape[1],hidden_units), 1e-5)
83
84     bias_o = np.zeros((1, 10))
85     bias_h = np.zeros((1, hidden_units))
86
87     weight_o, bias_o, weight_h, bias_h, accuracy, accuracy_valid, accuracy_test, loss, \
88         loss_valid, loss_test = learning(trainData, newtrain, W_o, v_o, W_h, v_h, epochs, \
89         gamma, learning_rate, bias_o, bias_h, validData, newvalid, testData, newtest)
90
91
92     hidden_out = relu(np.add(np.matmul(trainData, weight_h), bias_h))
93     prediction = softmax(np.add(np.matmul(hidden_out, weight_o), bias_o))
94     loss.append(CE(newtrain, prediction))
95     predict_result_matrix = np.argmax(prediction, axis = 1)
96     actural_result_matrix = np.argmax(newtrain, axis = 1)
97     compare = np.equal(predict_result_matrix, actural_result_matrix)
98     print("trainData accuracy: ", np.sum((compare==True))/(trainData.shape[0]))
99     accuracy.append(np.sum((compare==True))/(trainData.shape[0]))
100
101    hidden_input_valid = np.add(np.matmul(validData, weight_h), bias_h)
102    hidden_out_valid = relu(hidden_input_valid)
103    prediction_valid = softmax(np.add(np.matmul(hidden_out_valid, weight_o), bias_o))
104    loss_valid.append(CE(newvalid, prediction_valid))
105    predict_result_matrix_valid = np.argmax(prediction_valid, axis = 1)
106    actural_result_matrix_valid = np.argmax(newvalid, axis = 1)
107    compare_valid = np.equal(predict_result_matrix_valid, actural_result_matrix_valid)
108    print("validData accuracy: ", np.sum((compare_valid==True))/(validData.shape[0]))
109    accuracy_valid.append(np.sum((compare_valid==True))/(validData.shape[0]))
110
111    hidden_input_test = np.add(np.matmul(testData, weight_h), bias_h)
112    hidden_out_test = relu(hidden_input_test)
113    prediction_test = softmax(np.add(np.matmul(hidden_out_test, weight_o), bias_o))
114    loss_test.append(CE(newtest, prediction_test))
115    predict_result_matrix_test = np.argmax(prediction_test, axis = 1)
116    actural_result_matrix_test = np.argmax(newtest, axis = 1)
117    compare_test = np.equal(predict_result_matrix_test, actural_result_matrix_test)
118    print("testData accuracy: ", np.sum((compare_test==True))/(testData.shape[0]))
119    accuracy_test.append(np.sum((compare_test==True))/(testData.shape[0]))
120
121    iterations = range(len(accuracy))
122    plt.subplot(1, 2, 1)
123    plt.plot(iterations, accuracy)
124    plt.plot(iterations, accuracy_valid)
125    plt.plot(iterations, accuracy_test)
126    plt.ylabel('Accuracy')
127    plt.xlabel('Epochs')
128    # plt.legend(['train accuracy', 'valid accuracy', 'test accuracy'], loc='lower right')
129    plt.suptitle('Train, valid and test accuracy', fontsize=16)
130    plt.subplot(1, 2, 2)
131    plt.plot(iterations, loss)
132    plt.plot(iterations, loss_valid)
133    plt.plot(iterations, loss_test)
134    plt.ylabel('Loss')
```

```
    plt.xlabel('Epochs')
    plt.legend(['train loss', 'valid loss', 'test loss'], loc='upper right')
    plt.suptitle('Train, valid, test sets accuracy and loss', fontsize=16)
    plt.title
    plt.show()
```

Listing 10: Learning

1.4 Hyper-parameter Investigation

1.4.1 Number of hidden units

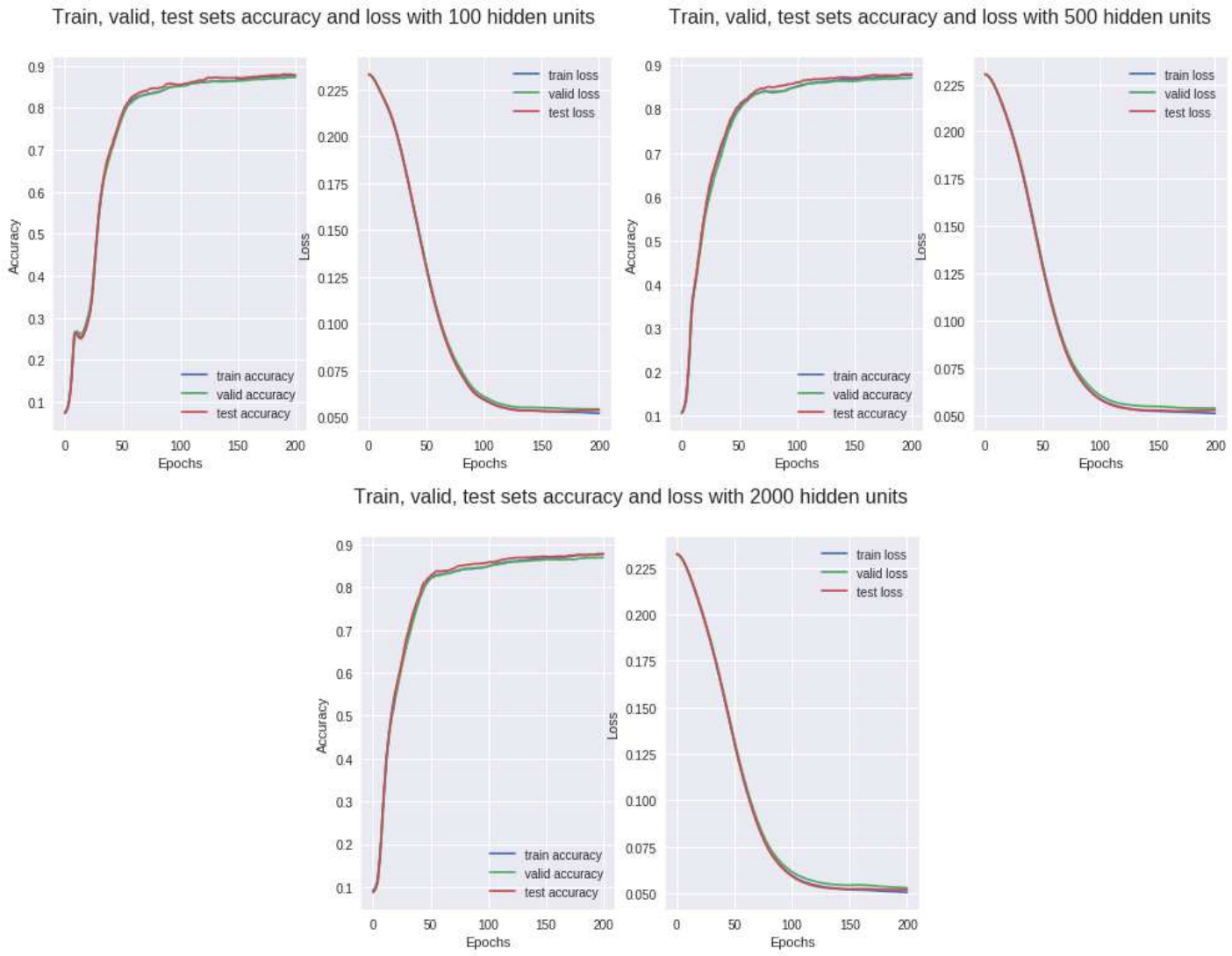

Figure 2: Accuracy and losses with different hidden units.

	Training data accuracy	Valid data accuracy	Test data accuracy	Training time
Hidden units = 100	87.31%	87.01%	87.81%	39.75s
Hidden units = 500	87.85%	87.53%	88.21%	136.81s
Hidden units = 2000	88.14%	87.53%	88.21%	444.40s

Table 2: Training, validation and test accuracy after training.

From table 2 above, we could see the accuracy for all three data sets have increased as the number of hidden units grow from 100 to 500. And from 500 to 2000, the time required for train the model was increasing largely, however, the accuracy for both validation data and test data remained the same, which means when the number of hidden units is around 500, the neural network obtained both high performance and high accuracy. One research suggested[1] the optimal number of

hidden units for first hidden layer can be calculated by $\sqrt{(m+2)N} + 2\sqrt{N/(m+2)}$, where N is the number of inputs, and m is the number of outputs, by using the formula, the optimal number of hidden units would be approximately 550 in this case, which is closer to 500 in table 2.

1.4.2 Early stopping

The early stopping point could be found when the accuracy of the validation data reached its highest value[2]. In the case of section 1.3, by training the model with 1000 hidden layers and 200 epochs, the highest accuracy for validation data occurred at the end of 200 epochs. The accuracy for each of three data sets are: 88.06% for training data, 87.45% for valid data, and 88.18% for test data. There was no early stopping point for 200 epochs, and even when the epochs increased to 300, the over fitting still didn't occur. The highest accuracy for valid data is at the last iteration, which are: 89.01% for training data, 88.28% for valid data, and 88.66% for test data.

2 Neural Networks in Tensorflow

2.1 Model implementation

For the neural network parameters, since the image is $28 \times 28 = 728$ and single channel, we define a input placeholder x, which has shape of $[None, 28, 28, 1]$. Here the row dimension is "None", because that we need the network to dynamically adapt our batch size. This tells placeholder that they will receive this dimension at the time when we will feed in the data to them. Similarly, for output placeholder, it has shape of $[None, n\_classes]$. Here $n\_classes = 10$.

As for weights and biases, we initialize them with the Xavier scheme. The first convolution layer has $32 - 4 \times 4$ filters, so the first key "wc1" in the weight dictionary has a shape of $[3, 3, 1, 32]$. The first and second numbers are filter size, the third is the number of channels in the input image, which is 1. And last represents the number of filters.

After applying convolution and max pooling operations, we are down-sampling the input image from $28 \times 28 \times 1$ to $14 \times 14 \times 1$. And we need to flatten this down-sampled output to feed the fully connected layer. And we also decided to double the output size of this layer. Hence, weight "wc2" has a shape of $[14 \times 14 \times 32, 64]$. Similarly, second fully connected layer has a shape of $[64, 128]$. Finally, the output layer of this neural network should have dimension equals to previous layer output channels times the number of classes.

Same principle for biases initialization.

```
x = tf.placeholder("float", [None, 28,28,1])
y = tf.placeholder("float", [None, n_classes])
weights = {
    'wc1': tf.get_variable('W0', shape=(3,3,1,32), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'wc2': tf.get_variable('W1', shape=(14*14*32,64), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'wc3': tf.get_variable('W3', shape=(64,128), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'out': tf.get_variable('W6', shape=(128,n_classes), \
        initializer=tf.contrib.layers.xavier_initializer()),
}
biases = {
    'bc1': tf.get_variable('B0', shape=(32), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'bc2': tf.get_variable('B1', shape=(64), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'bc3': tf.get_variable('B2', shape=(128), \
        initializer=tf.contrib.layers.xavier_initializer()),
    'out': tf.get_variable('B4', shape=(10), \
        initializer=tf.contrib.layers.xavier_initializer()),
}
```

Listing 11: Weights and bias initialization

Two functions below are convolution and max-pooling layers. And convolution layer has ReLU activation function. In the *conv2d()* function we passed 4 arguments: input x, weights W, bias b and strides. This last argument is by requirement set to 1,

```
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], \
```

```
        padding='SAME', use_cudnn_on_gpu=True)
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

def maxpool2d(x, k=2):
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], \
        strides=[1, k, k, 1],padding='SAME')
```

Listing 12: Convolutional layer functions

The *conv_net()* function takes 3 arguments as an input: the input x and the weights and biases dictionaries. This function build up the architecture of our convolutional neural network. Detailed layer interpretation is documented in the code below.

At this point, we can start with constructing the training model by call the *conv_net()* function. Since this is a multi-class classification problem, we are going to use softmax activation on the output layer as implemented in line 42 of below code.

```
def conv_net(x, weights, biases):
    # 1. Input Layer is x.
    epsilon = 1e-3

    # 2. A 4    4 convolutional layer, with 32 filters, using vertical and horizontal strides of 1.
    # here we call the conv2d function we had defined above and pass the input image x, weights wc1 and
     bias bc1.
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])

    # 3. ReLU activation
    # ReLU_conv1 = fc1 = tf.nn.relu(conv1)

    # 4. A batch normalization layer
    batch_mean2, batch_var2 = tf.nn.moments(conv1, [0, 1, 2])
    scale2 = tf.Variable(tf.ones([32]))
    beta2 = tf.Variable(tf.zeros([32]))
    batch_norm = tf.nn.batch_normalization(conv1,batch_mean2,batch_var2,beta2,scale2,epsilon)

    # 5. A max 2    2 max pooling layer.
    # Max Pooling (down-sampling), this chooses the max value from a 2*2 matrix window and outputs a
    14*14 matrix.
    pool_conv1 = maxpool2d(batch_norm, k=2)

    # 6. Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(pool_conv1, [-1, weights['wc2'].get_shape().as_list()[0]])

    # 7. Fully connected layer
    fc1 = tf.add(tf.matmul(fc1, weights['wc2']), biases['bc2'])

    # 7. ReLU activation
    fc1 = tf.nn.relu(fc1)

    # 8. Fully connected layer
    fc2 = tf.reshape(fc1, [-1, weights['wc3'].get_shape().as_list()[0]])
    fc2 = tf.add(tf.matmul(fc2, weights['wc3']), biases['bc3'])

    # 9. Outputlayer
    out = tf.add(tf.matmul(fc2, weights['out']), biases['out'])
    return out


pred = conv_net(x, weights, biases)

# 10. Cross Entropy loss
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Here you check whether the index of the maximum value of the predicted image
# is equal to the actual labelled image. and both will be a column vector.
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))

# Calculate accuracy across all the given images and average them out.
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

```

```
# Initializing the variables
init = tf.global_variables_initializer()
```

Listing 13: Neural network architecture

Finally, we can training and testing the model. Define a loop to run number of training iterations that we specified, in this case is 50 epochs. Right after that, we initiate a second for loop, which is for the number of batches. After each training iteration is completed, shuffle the training data.

```
with tf.Session() as sess:
    sess.run(init)
    train_loss = []
    valid_losses = []
    test_losses = []
    train_accuracy = []
    valid_accuracy = []
    test_accuracy = []

    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    trainData = trainData.reshape(-1, 28, 28, 1)
    validData = validData.reshape(-1, 28, 28, 1)
    testData = testData.reshape(-1, 28, 28, 1)
    summary_writer = tf.summary.FileWriter('./Output', sess.graph)

    for i in range(training_iters):
        trainData, trainTarget = shuffle(trainData, trainTarget)
        newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)
        for batch in range(len(trainData)//batch_size):
            batch_x = trainData[batch*batch_size:min((batch+1)*batch_size,len(trainData))]
            batch_y = newtrain[batch*batch_size:min((batch+1)*batch_size,len(newtrain))]
            # Run optimization op (backprop).
            opt = sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        loss, acc = sess.run([cost, accuracy], feed_dict={x: trainData, y: newtrain})
        print("Iter " + str(i) + ", Loss= " + \
                      "{:.6f}".format(loss) + ", Training Accuracy= " + \
                      "{:.5f}".format(acc))
        print("Optimization Finished!")
        # Calculate accuracy for all 10000 mnist test images
        test_acc,test_loss = sess.run([accuracy,cost], feed_dict={x: testData,y : newtest})
        valid_acc,valid_loss = sess.run([accuracy,cost], feed_dict={x: validData,y : newvalid})
        train_loss.append(loss)
        valid_losses.append(valid_loss)
        test_losses.append(test_loss)
        train_accuracy.append(acc)
        test_accuracy.append(test_acc)
        valid_accuracy.append(valid_acc)
        print("Testing Accuracy:","{:.5f}".format(test_acc))
    summary_writer.close()
```

Listing 14: Train session

2.2 Model Training

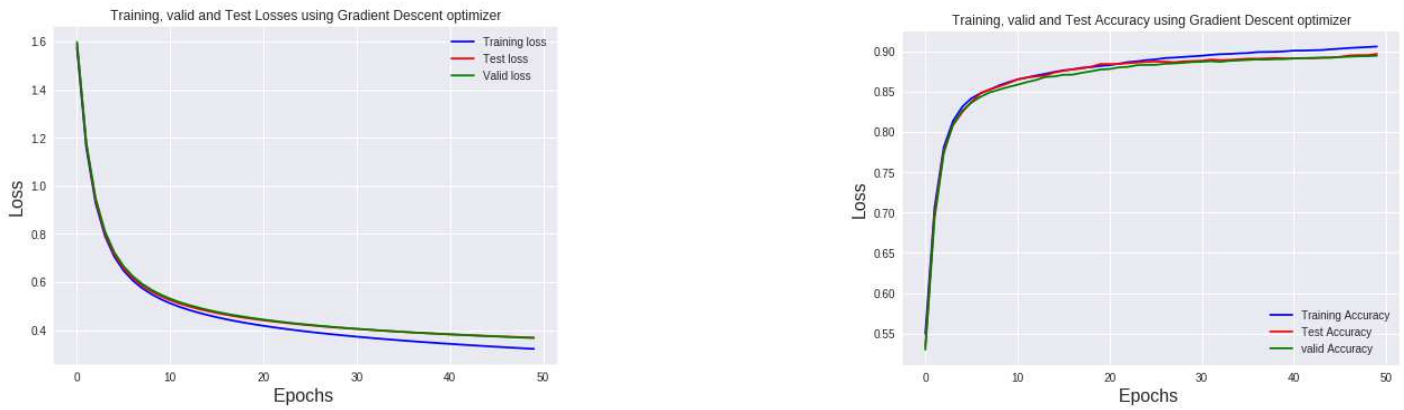

Figure 3: Accuracy and losses of CNN using SGD optimizer.

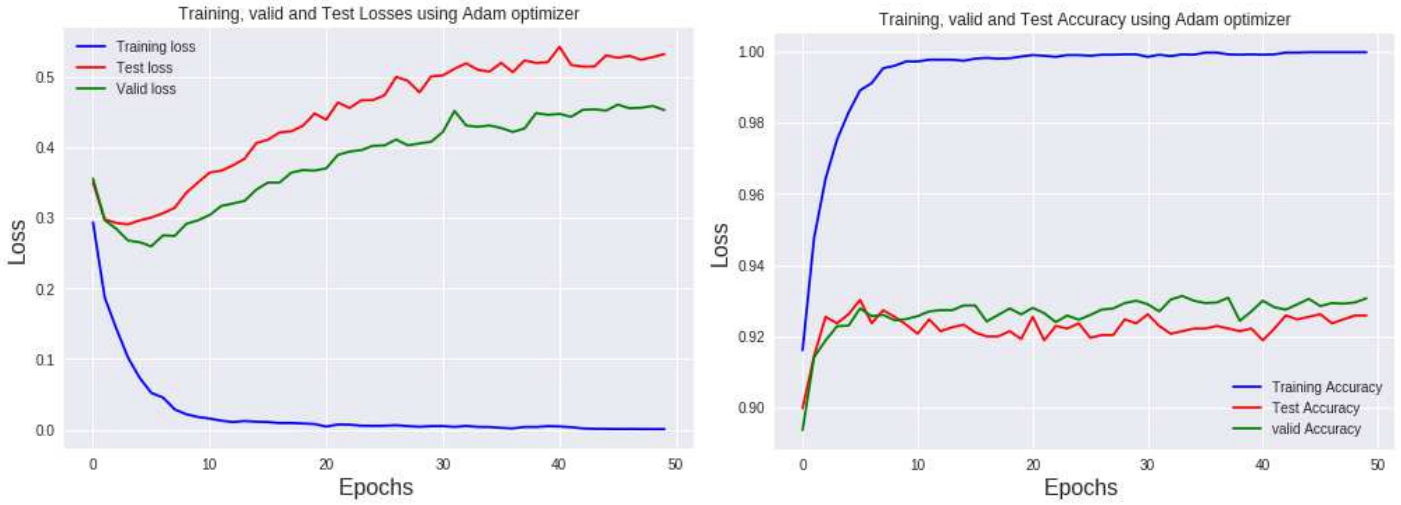

Figure 4: Accuracy and losses of CNN using Adam optimizer.

2.3 Hyper-parameter Investigation

Notice that previous part, when we train the model using Adam Optimizer, a significant over-fitting occurred. But it is not the case for SGD. Therefore, we are not going to consider SGD in this part.

2.3.1 L2 Normalization

λ	Training Accuracy	Validation Accuracy	Testing Accuracy
0	99.97%	93.13%	93.06%
0.01	99.61%	93.41%	92.58%
0.1	93.12%	91.43%	91.59%
0.5	87.58%	86.45%	87.37%

Table 3: Final training, validation and test accuracy for different regularization

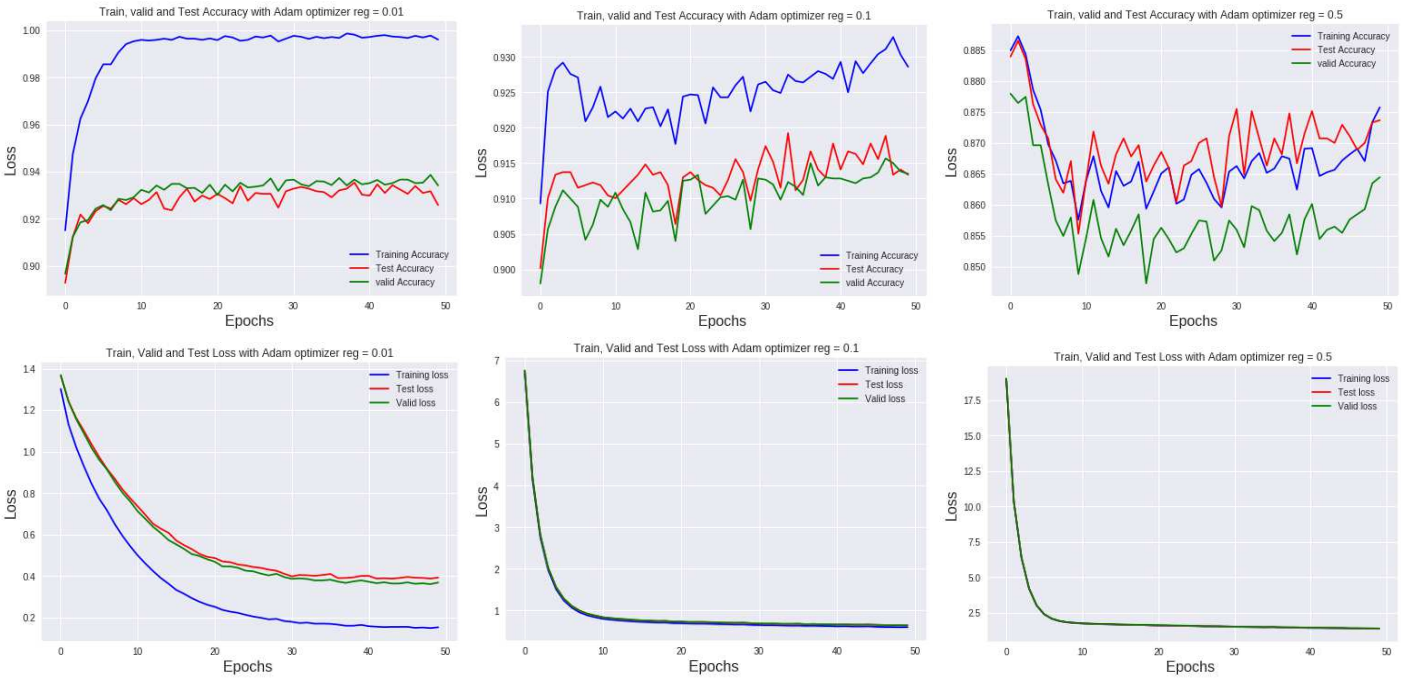

Figure 5: Training, validation and test accuracy and loss curves for different regularization.

Adding L2 regularization is one way to prevent the model from over fitting the training data, by changing the decay coefficient λ, the accuracy of the training model changes. As shown in table 3, as λ increased, the accuracy of all three

data sets decreased, since L2 used squares, it already had emphasized the error, and in addition, the λ parameter which is directly proportional to the penalty is also stronger the penalty as the coefficient increased, therefore, as the penalty (decay coefficient) increased, there will be more noise added to the model, it did prevent the model from over fitting, however, it also leads to a lower accuracy.

2.3.2 Dropout

Deprecated	Training Accuracy	Validation Accuracy	Testing Accuracy
0	99.97%	93.13%	93.06%
0.9	99.82%	92.35%	92.62%
0.75	99.43%	92.33%	92.03%
0.5	98.56%	91.87%	91.63%

Table 4: Final training, validation and test accuracy for different dropout rates

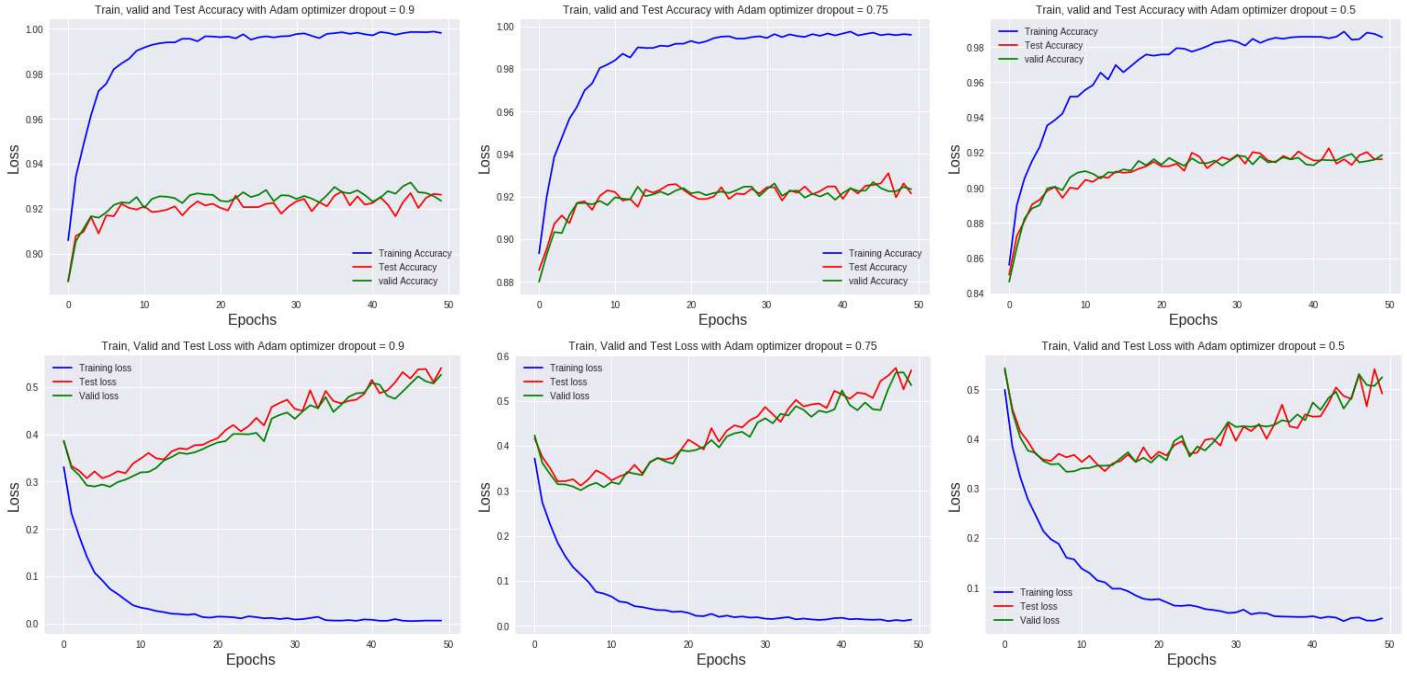

Figure 6: Training, validation and test accuracy and loss curves for different regularization.

Another way to control model from over fitting is to add dropouts to certain layers. As shown in table 4 above, the "Deprecated" column had shown the probability of each hidden unit that will not be dropped out, such as 0.9 means there are a 10% chance for each hidden unit to be set to 0 (drop out). As the probability of dropout unit increased, more noise were added to the training model, therefore, the accuracy for all three data sets had decreased.

References

[1] "Learning capability and storage capacity of two-hidden-layer feedforward networks," *IEEE Transactions on Neural Networks*, vol. 14, pp. 274–281, March 2003.

[2] L. Prechelt, "Early stopping - but when?," in *Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2*, pp. 55–69, Springer-Verlag, 1997.